

real-time automata

Henning Dierks¹

University of Oldenburg, Fachbereich Informatik, Postfach 2503, 2900 Oldenburg, Germany

Abstract

We introduce PLC-automata as a new class of automata which are tailored to deal with real-time properties of programmable logic controllers (PLCs). These devices are often used in industrial practice to solve controlling problems. Nevertheless, PLC-automata are not restricted to PLCs, but can be seen as a model for all polling systems. A semantics in an appropriate real-time temporal logic (duration calculus) is given and an implementation schema that fits the semantics is presented in a programming language for PLCs. A case study is used to demonstrate the suitability of this approach. We define several parallel composition operators, and present an alternative semantics in terms of timed automata for which model-checkers are available. © 2001 Elsevier Science B.V. All rights reserved.

Keywords: Real time; Specification; Formal methods; Duration calculus; PLC

1. Introduction

In this paper we propose a language to specify real-time systems that fits both the needs of computer scientists and programmers of such systems. Formal specification and verification of real-time systems that are used in practice depend on the communication between the scientist who models the behaviour of the system by formal methods and the programmer who is working in practice with it.

This language which we call “PLC-automata” is motivated by the experiences we made in the UniForM-project [21] with an industrial partner. The aim of the project is the development of real-time systems in a workbench using combinations of formal methods. We present a formal semantics that allows formal reasoning and proving

¹ This research was supported by the German Ministry for Education and Research (BMBF) as part of the project UniForM under Grant No. FKZ 01 IS 521 B3 and by the Leibniz Programme of the German Research Council (DFG) under Grant OI 98/1-1.

E-mail address: dierks@informatik.uni-oldenburg.de (H. Dierks).

correctness using the duration calculus [33] as semantic basis. We also give an implementation of such systems in a particular hardware called programmable logic controllers (PLC).

These PLCs are very often used in practice to implement real-time systems. The reason is that they provide both an automatic polling mechanism and convenient methods to deal with time by explicit timers in their programming languages. Nevertheless, every computer system can be used to implement the proposed language if a comparable handling of time and an explicit polling is added.

Furthermore, the language can be regarded as a definition of a small but implementable subset of timed automata [2]. See Section 10 for details where this formalism is used to define an operational semantics for PLC-automata. The main difference between both approaches is the polling concept. Another difference is that PLC-automata assume an asynchronous way to react to inputs while timed automata react synchronously to inputs.

2. The behaviour of programmable logic controllers

Programmable logic controllers (PLC) are often used in industry for solving tasks calling for real-time problems like railway crossings, traffic control, or production cells. Due to this special application background PLCs have features for making the design of time- and safety-critical systems easier:

- PLCs have input and output channels where sensors and actuators respectively can be plugged in.
- They behave in a cyclic manner where every cycle consists of the following phases:
 - Polling all inputs and storing the read values.
 - Computing the new values for the outputs.
 - Updating all outputs.

The repeated execution of this cycle is managed by the operating system. The only part the programmer has to adapt is the computing phase. Thus, PLCs are implemented polling machines realising the typical method of solving time-critical problems in reality.

- Depending on the program and on the number of inputs and outputs there is an upper time bound for a cycle that can be used to calculate the reaction time.
- Convenient standardised libraries are given to simplify the handling of time.

Although these characteristics are quite useful, PLC-programmers have to face the following problem: If an input signal does not hold for at least the maximum amount of time needed for a cycle, one cannot be sure that the PLC will ever read this signal. This problem can be solved either by

- changing the sensors used in the setting or by
- using PLCs that are fast enough.

The decision in which way the problem should be solved depends on availability and costs of both faster PLCs and sensors that assure longer lasting signals.

Another important feature of PLCs is that they can be coupled: the output of one PLC can be the input of another PLC. In fact, their operating systems do not differentiate between a sensor's input and a PLC's input and between an output to actuators or to PLCs, respectively. Thus, the programmer is again obliged to consider how long an output signal from one PLC will be held and how long it must be held to make sure that it has been noticed by the other PLC. In physically distributed applications several busses are in use to connect PLCs. They introduce delays which have to be considered in the parallel composition of PLCs.

Note that these considerations are an advantage of using PLCs. They oblige the programmer to check both the sensor and cycle time, which makes the assumptions concerning the hardware explicit.

3. The definition of PLC-automata

In this section we propose a formalism which is designed for both the needs of computer scientists and of engineers programming PLCs. Engineers, often being electrical engineers, are used to develop PLC-programs in assembler-like languages or languages that are closely related to circuit diagrams.

In the UniForM-project [21] we made the experience that automata-like pictures can serve as a common basis for computer scientists and engineers because the engineers gave them a semantics suitable to PLCs in an intuitive way. This was the motivation for us to formalise these pictures and to define a formal semantics for them in a suitable temporal logic. On the one hand, this allows formal reasoning; on the other hand, this respects the behaviour of PLCs and the intuitive semantics given by the programmers.

In a railway case study of the UniForM-project we are dealing with problems like the following one:

Example 1. Consider a train detecting sensor that signals “*tr*” (train) if a train is approaching and “*no_tr*” (no train) if not. Unfortunately, the sensor can stutter for up to 4 s after a train has passed the sensor. Assume that the temporal distance between two subsequent trains is at least 6 s. Develop a system that filters the stuttering.

The automaton in Fig. 1 shows a PLC-Automaton representation of such a device. The automaton consists of two states “N” and “T”. It reacts on the filter's input *no_tr* or *tr* accordingly. The behaviour of the PLC during a cycle is described by the transition relation of the automaton. In Fig. 2 a run of a PLC is given according to the automaton in Fig. 1. This automaton should behave as follows:

- It starts in state “N”.
- If it is in state “N” and the input is “*tr*”, it changes to state “T”. Otherwise it remains in “N”.
- In “T” the automaton holds this state for 5 s. Afterwards it remains in this state as long as polling the input yields a “*tr*”. Otherwise it changes to state “N” and continues as before.

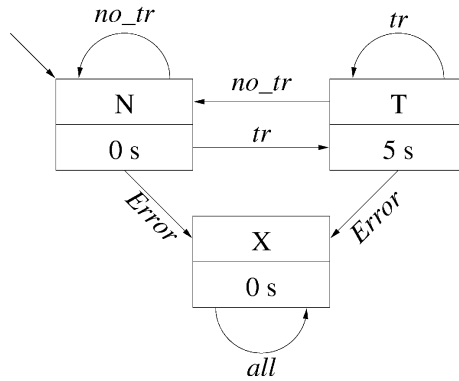


Fig. 3. Filtering device with detection of errors.

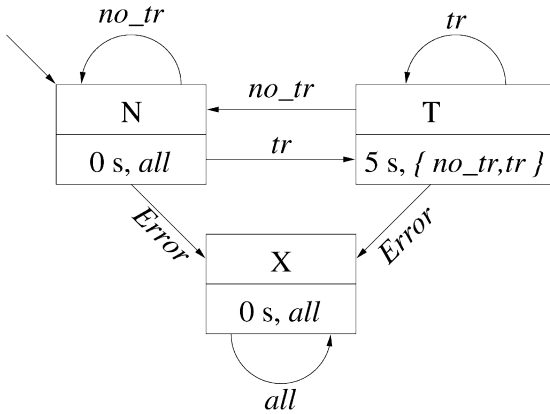


Fig. 4. Filtering device with immediate detection of errors.

after a change from “N” to “T”. In this case the automaton would not change to “X” immediately because it is required to stay for at least 5 s in “T”.

To solve this problem we introduce a set of inputs for each state of the automaton for a special treatment: The informal meaning of a state equipped with a delay time t and a set A of inputs is that inputs contained in A are ignored for the first t s staying in this state. Inputs outside A are never ignored, i.e. they force the automaton to react immediately. Fig. 4 shows how this extension can be used to solve Example 2. It behaves the same way as the automaton of Fig. 1 provided that no “Error” signal occurs. If an “Error” occurs, the automaton of Fig. 4 changes to state “X” regardless in which state it was before and how long it was there.

In summary, we define an automaton-like structure extended by some components:

Definition 3. A tuple $\mathcal{A} = (Q, \Sigma, \delta, q_0, \varepsilon, S_t, S_e, \Omega, \omega)$ is a *PLC-automaton* if

- Q is a nonempty, finite set of *states*,

- Σ is a nonempty, finite set of *inputs*,
- δ is a *transition function* of type $Q \times \Sigma \rightarrow Q$,
- $q_0 \in Q$ is the *initial state*,
- $\varepsilon > 0$ is the *upper bound* for a cycle.
- S_t is a function of type $Q \rightarrow \mathbb{R}_{\geq 0}$ assigning to each state q a *delay time* how long the inputs contained in $S_e(q)$ should be ignored,
- S_e is a function of type $Q \rightarrow \mathcal{P}(\Sigma)$ assigning to each state q a set of *delayed inputs* that cause no change of the state during the first $S_t(q)$ seconds the automaton stays in this state,
- Ω is a nonempty, finite set of *outputs*, and
- ω is an *output function* of type $Q \rightarrow \Omega$.

The additional components are needed to model PLC-behaviour and to enrich the language for dealing with real-time aspects. The ε represents the upper bound for a cycle of a PLC and enables us to model this cycle in the semantics. The functions S_t and S_e attach to each state of \mathcal{A} a delay time and a set of inputs. We want the automaton to remain in state q for at least $S_t(q)$ seconds (“ t ” stands for “time delay”) provided that only inputs in $S_e(q)$ are read (“ e ” stands for “expected inputs”). E.g. in Fig. 4 we want the system to hold output “T” for at least 5 s provided that only inputs in $\{no_tr, tr\}$ are read. In other words, inputs in $S_e(q)$ are ignored for the first $S_t(q)$ seconds.

An equivalent description is that the state q is held for $S_t(q)$ seconds, but if during this period an input in $\Sigma \setminus S_e(q)$ is *read* the automaton will react within one cycle. Note that an input lasting only very shortly need not to be noticed. That means that an input can either hold and be read (e.g. the second “ no_tr ” in Fig. 2) or hold shortly and not be read (e.g. the first “ tr ” in Fig. 2).

PLC-automata look similar to timed automata [2] but the details are different. In our approach we deal with reaction times; this is made precise in the semantics defined in Section 5. In Section 10 we will give an alternative timed automaton semantics for PLC-automata and in this semantics we have to make the asynchronous behaviour of PLC-Automata and the reaction times explicit, because timed automata represent a synchronous approach without reaction times.

4. The duration calculus

In this paper we use duration calculus (abbreviated DC), a dense time interval temporal logic developed by Zhou Chaochen and others [33, 27, 16], as the predicate language to describe properties of real-time systems. This choice is mostly motivated by our previous experience and acquired fluency in this logic, but also by the convenience with which the interval and continuous time aspects of DC allow us to express and reason about reaction times of components and durations of states.

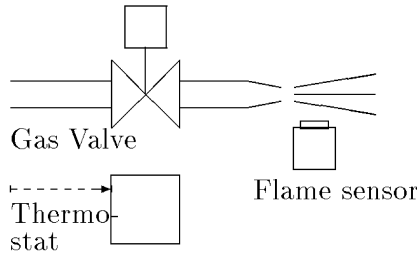


Fig. 5. The gas burner.

4.1. Motivation

We consider the gas burner case study of the ProCoS-project to illustrate the usage of DC as high-level specification language. The gas burner [29] is triggered by a thermostat; it can directly control a gas valve and monitor the flame (Fig. 5).

This physical system is modelled by three Boolean observables: “hr” (heatrequest) represents the state of the thermostat, “fl” (flame) represents the presence of a flame at the gas valve, “gas” represents the state of the gas valve. One of the top-level requirements is that

in every period shorter than 30 s gas must not leak for more than 4 s.

This is expressed by the following DC-formula:

$$\Box \left(\ell \leq 30 \Rightarrow \int (\text{gas} \wedge \neg \text{fl}) \leq 4 \right) \quad (1)$$

Here the \int -operator accumulates all durations of leaks (modelled by the assertion $\text{gas} \wedge \neg \text{fl}$) over a given interval. Hence, (1) can be read as follows: For every interval (\Box) of a length of at most 30 s ($\ell \leq 30$) the sum of all leak durations within that interval is at most 4 s ($\int (\text{gas} \wedge \neg \text{fl}) \leq 4$).

The \int -operator is the main advantage of the DC because it allows to reason about the *sum of specific durations* which is not possible in other real-time logics like TCTL for timed automata [1]. With this operator it is not difficult to specify properties like quasi-fairness or mutual exclusion of processes. For example, quasi-fairness of two processes P_1 and P_2 which are to enter a critical section cs_1 and cs_2 , is specified by

$$\Box \left(\ell \geq 10 \Rightarrow \left| \int (P_1 = cs_1) - \int (P_2 = cs_2) \right| \leq \frac{\ell}{10} \right).$$

This formula forbids that one process occupies its critical section substantially longer (i.e. 10%) than the other one during an interval of at least 10 s. It says that in every interval (\Box) of at least 10 s ($\ell \geq 10$) the distance ($|\dots - \dots|$) between the occupation times of P_1 and P_2 ($\int P_i = cs_i$, $i = 1, 2$) is at most ten percent of the time ($\leq \ell/10$).

Mutual exclusion is simply specified by

$$\int (P_1 = cs_1 \wedge P_2 = cs_2) = 0$$

which means that the accumulated durations of the simultaneous occupations is 0. Note that this formula means that P_1 and P_2 are in their critical section simultaneously for at most finitely many points in every interval. Due to the integration finitely many points do not play a role for the validity of a DC-formula.

4.2. Syntax

Formally, the syntax of duration calculus distinguishes *terms*, *duration terms* and *duration formulae*. *Terms* τ have a certain type and are built from time-dependent observables *obs* like gas or fl, *rigid variables* x representing time-independent variables, and are closed under typed operators *op*:

$$\tau ::= obs \mid x \mid op(\bar{\tau})$$

where $\bar{\tau}$ is a vector of terms. Note that *op* include nullary operators, i.e., constants. Terms of Boolean type are called *state assertions*. We use S , P and occasionally Q for a typical state assertion.

Duration terms θ are of type real but their values depend on a given time interval. The simplest duration term is the symbol ℓ denoting the *length* of the given interval. The name duration calculus stems from the fact that for each state assertion S there is a duration term $\int S$ measuring the *duration* of S , i.e. the accumulated time S holds in the given interval. Formally,

$$\theta ::= \ell \mid \int S \mid op_{real}(\bar{\theta})$$

where op_{real} is an real-valued operator and $\bar{\theta}$ a vector of duration terms.

Duration formulae denote truth values depending on a given time interval. They are built from relations *rel* applied to duration terms, and are closed under the *chop operator* (denoted by “;”), propositional connectives op_{Bool} , and quantification $Q \in \{\forall, \exists\}$ over rigid variables x . We use F for a typical duration formula:

$$F ::= rel(\bar{\theta}) \mid F_1; F_2 \mid op_{Bool}(\bar{F}) \mid Qx.F$$

where \bar{F} is a vector of duration formulae.

4.3. Semantics

The semantics of duration calculus is based on an *interpretation* \mathcal{I} that assigns a fixed meaning to each observable, rigid variable and operator symbol of the language. To an observable *obs* the interpretation \mathcal{I} assigns a function

$$obs_{\mathcal{I}} : \text{Time} \rightarrow D_{obs}$$

with $\text{Time} = \mathbb{R}_{\geq 0}$. This induces inductively the semantics of terms and hence state assertions. For a state assertion S it is a function

$$S_{\mathcal{I}} : \text{Time} \rightarrow Bool$$

where *Bool* is identified with the set $\{0, 1\}$.

The semantics of a duration term θ is denoted by $\mathcal{J}(\theta)$ and yields a real value depending on a given time interval $[b, e] \subseteq \text{Time}$. In particular, ℓ denotes the length of $[b, e]$ and $\int S$ the duration of the state assertion S in $[b, e]$ as given by the integral. Formally,

$$\begin{aligned}\mathcal{J}(\ell)[b, e] &= e - b, \\ \mathcal{J}\left(\int S\right)[b, e] &= \int_b^e S_{\mathcal{J}}(t) dt.\end{aligned}$$

The semantics of a duration formula F denotes a truth value depending on \mathcal{J} and a given time interval $[b, e]$. We write $\mathcal{J}, [b, e] \models F$ if that truth value is *true* for \mathcal{J} and $[b, e]$. The definition is by induction on the structure of F . The cases of relations, propositional connectives and quantification are handled as usual. For example, $\mathcal{J}, [b, e] \models \int S \leq k$ if the duration $\int_b^e S_{\mathcal{J}}(t) dt$ is at most k . For $F_1; F_2$ (read as F_1 chop F_2) we define $\mathcal{J}, [b, e] \models F_1; F_2$ if the interval $[b, e]$ can be “chopped” into two subintervals $[b, m]$ and $[m, e]$ such that $\mathcal{J}, [b, m] \models F_1$ and $\mathcal{J}, [m, e] \models F_2$.

Since the initial values of observations are important in our application, we especially consider time intervals starting at time 0 and define: a duration formula F *holds* in an interpretation \mathcal{J} if $\mathcal{J}, [0, t] \models F$ for all $t \in \text{Time}$. Formal requirements are specified by a number of suitable duration formulae and considers all interpretations for which the conjunction of the DC-formulae holds.

4.4. Abbreviations and precedence rules

Besides this basic syntax various abbreviations are usual in DC:

$$\begin{aligned}\text{true :} & \quad \text{true} \stackrel{\text{df}}{=} \ell \geq 0 \\ \text{false :} & \quad \text{false} \stackrel{\text{df}}{=} \neg \text{true} \\ \text{point interval :} & \quad \square \stackrel{\text{df}}{=} \ell = 0 \\ \text{everywhere :} & \quad [P] \stackrel{\text{df}}{=} \int P = \ell \wedge \ell > 0 \\ \text{somewhere :} & \quad \diamond F \stackrel{\text{df}}{=} \text{true}; F; \text{true} \\ \text{always :} & \quad \square F \stackrel{\text{df}}{=} \neg \diamond \neg F \\ & \quad F^t \stackrel{\text{df}}{=} (F \wedge \ell = t) \\ & \quad F^{\sim t} \stackrel{\text{df}}{=} (F \wedge \ell \sim t) \text{ with } \sim \in \{<, \leq, >, \geq\}\end{aligned}$$

The following so-called *standard forms* are often used because they are useful to describe dynamic behaviour:

$$\begin{aligned}\text{followed-by:} & \quad F \rightarrow [P] \stackrel{\text{df}}{=} \square \neg (F; [\neg P]) \\ \text{timed leads-to:} & \quad F \xrightarrow{t} [P] \stackrel{\text{df}}{=} (F^t) \rightarrow [P] \\ \text{timed up-to:} & \quad F \xrightarrow{\leq t} [P] \stackrel{\text{df}}{=} (F^{\leq t}) \rightarrow [P]\end{aligned}$$

As before we have $t \in \text{Time}$. Intuitively, $F \rightarrow [P]$ expresses the fact that whenever a pattern given by a formula F is observed, then it will be “followed by” an interval

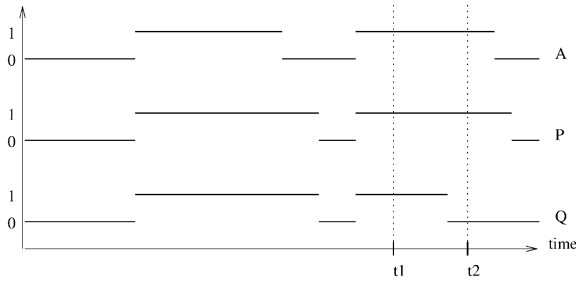


Fig. 6. Example for “followed by”.

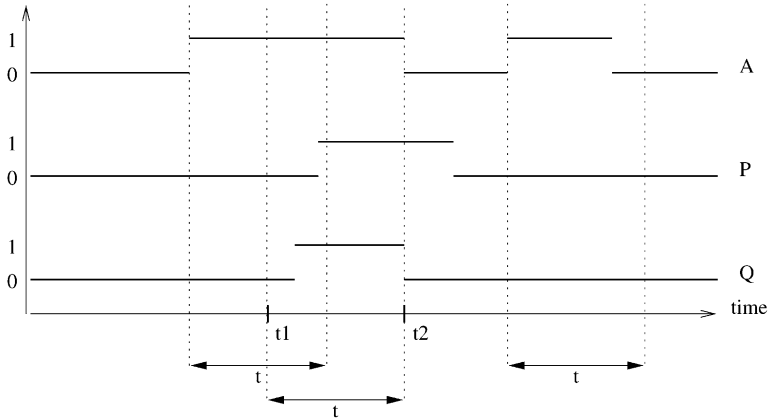


Fig. 7. Example for “leads to”.

in which P holds. Fig. 6 exhibits an interpretation for the boolean observables A , P and Q . In this interpretation $[A] \rightarrow [P]$ is true because for every interval in which $[A]$ holds there is a *immediately* succeeding interval in which $[P]$ is true. $[A] \rightarrow [Q]$ is not true: choosing the interval $[t1, t2]$ there is no *immediately* succeeding interval in which $[Q]$ holds.

In the “leads-to” form this pattern is required to have a length t (Fig. 7), and in the “up-to” form the pattern is bounded by a length “up to” t . Note that the “leads-to” does not simply say that whenever F holds then t time units later $[P]$ holds; rather, a stability of F for t time units is required before we can be sure that $[P]$ holds. Fig. 7 demonstrates the meaning of \xrightarrow{t} . Above there are two phases where $[A]$ is true and the former phase lasts longer than t seconds. $[A] \xrightarrow{t} [P]$ holds because for every interval of length t in which $[A]$ holds there is a succeeding interval in which $[P]$ is true. $[A] \xrightarrow{t} [Q]$ is not true: choosing the interval $[t1, t2]$ there is no succeeding interval in which $[Q]$ holds.

The “up-to” form is mainly used to specify certain stability conditions. For example $[\neg P]; [P] \xrightarrow{\leq t} [P]$ is an expression that is true iff P is stable for at least t seconds whenever P becomes true.

To save parentheses the following precedence rules are used:

- (1) \int
- (2) real operators
- (3) real predicates
- (4) \neg , \square , \diamond
- (5) $;$
- (6) \wedge , \vee
- (7) \Rightarrow , \rightarrow , $\xrightarrow{\leq t}$, \xrightarrow{t}
- (8) quantification

5. A duration calculus semantics for PLC-automata

In this section we define the semantics of the PLC-automata proposed in Section 3 with DC-formulae. This enables us to prove real-time properties of such automata by means of logical reasoning.

The semantics $\llbracket \mathcal{A} \rrbracket_{\text{DC}}$ of a PLC-automaton $\mathcal{A} = (Q, \Sigma, \delta, q_0, \varepsilon, S_e, S_t, \Omega, \omega)$ is given by the conjunction of the following predicates regarding the observables $\mathcal{I}_{\mathcal{A}} : \text{Time} \rightarrow Q$, $\mathcal{J}_{\mathcal{A}} : \text{Time} \rightarrow \Sigma$ and $\mathcal{O}_{\mathcal{A}} : \text{Time} \rightarrow \Omega$. First of all, the starting of the automaton in the proper initial state is expressed by

$$[\square \vee [q_0]; \text{true}], \quad (2)$$

where we use the convention that $[q_0]$ is an abbreviation of $[\mathcal{I}_{\mathcal{A}} = q_0]$. Next, we want to describe the behaviour of the automaton in a state q . The cyclic behaviour of PLCs has to be reflected in the semantics to achieve a realistic modelling. One question the semantics should answer is: When a state q is entered, what kind of input can influence the behaviour of the PLC? The answer to this question is:

- only the inputs after entering q and,
- only the inputs during the last cycle-time ε .

This is expressed by the following predicates where A ranges over all sets of inputs with $\emptyset \neq A \subseteq \Sigma$. In the formulae we use A as an abbreviation for $\mathcal{J}_{\mathcal{A}} \in A$ and $\delta(q, A)$ for $\mathcal{I}_{\mathcal{A}} \in \{\delta(q, a) \mid a \in A\}$, respectively:

$$[\neg q]; [q \wedge A] \rightarrow [q \vee \delta(q, A)] \quad (3)$$

$$[q \wedge A] \xrightarrow{\varepsilon} [q \vee \delta(q, A)] \quad (4)$$

Statement (3) formalises the fact that after a change of the automaton's state to q , only the set of inputs A that is valid after the change can have an effect on the behaviour in the future. Statement (4) represents the formalisation of the cyclic behaviour of PLCs. A PLC reacts only to inputs that occurred during the last cycle. Preceding inputs are forgotten and cannot influence the behaviour of the PLC-automaton anymore.

The quantification over all nonempty subsets of the input alphabet was motivated by the behaviour of the PLCs. The more we know about the inputs during the last

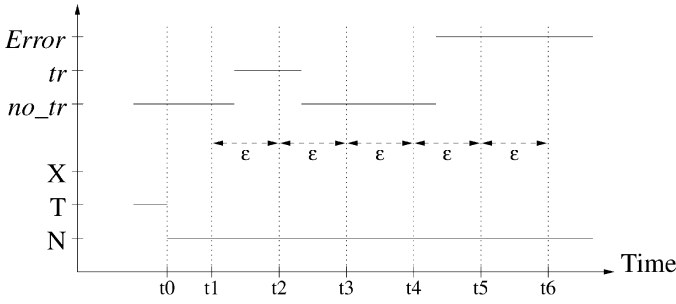


Fig. 8.

cycle the more we know about the actions of the PLC. For example, it is necessary that an input a is held for at least ε seconds to assure that the PLC can only react to this input. This is directly reflected in the semantics as well. If there is an interval of length ε , predicate (4) can be applied to this interval with $A = \{a\}$. Consequently, after this interval only transitions with label a are allowed.

Fig. 8 exhibits a possible history of the PLC-automaton in Fig. 4. The application of statement (3) needs two time points. The first point is a change to a state q of the automaton. In Fig. 8 the system changes from T to N at time t_0 . The second time point is later than the first and requires the state to be constant between both time points. Predicate (3) assures that after the second time point there is an interval in which the state is either q or a state $\delta(q, a)$ where a is an input that was valid between both time points.

Hence, the application of (3) to the history of Fig. 8 yields the following results: At t_1 the state can remain in N only. After t_2 , t_3 , and t_4 one knows that only changes to T are possible. At t_5 and t_6 no change is forbidden by (3).

For the application of (4) two time points are needed, too. The distance between both points has to be ε seconds and the state has to be constant in between. Due to (4) we know again that after the second time point the state is either q or a state $\delta(q, a)$ where a is an input that was valid between both time points.

The application of (4) to Fig. 8 now gives us the information that

- after t_2 and t_3 the state remains N or changes to T ,
- after t_4 the state is not allowed to change, and
- after t_5 and t_6 changes to T are forbidden.

Note that (4) cannot be applied in such a way that we gain information at t_1 because the state changed in less than ε seconds before t_1 .

For states without a stability requirement we expect a change to $\delta(q, a)$ in at most 2ε seconds. For states with a stability requirement we expect this behaviour after the required period of time. This leads us to additional statements in the semantics:

$$S_i(q) = 0 \wedge q \notin \delta(q, A) \Rightarrow \Box(\lceil q \wedge A \rceil \Rightarrow \ell < 2\varepsilon) \quad (5)$$

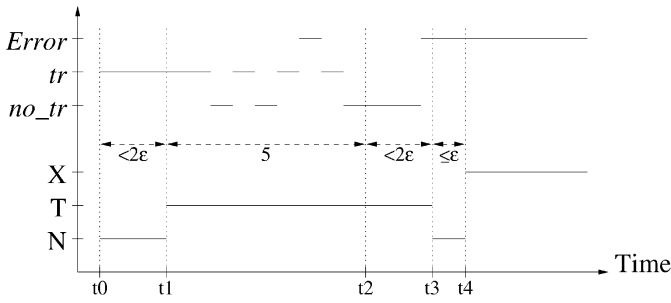


Fig. 9.

$$S_t(q) > 0 \wedge q \notin \delta(q, A) \Rightarrow \Box(\lceil q \rceil^{S_t(q)}; \lceil q \wedge A \rceil \Rightarrow \ell < S_t(q) + 2\varepsilon) \quad (6)$$

$$S_t(q) = 0 \wedge q \notin \delta(q, A) \Rightarrow \lceil \neg q \rceil; \lceil q \wedge A \rceil^\varepsilon \rightarrow \lceil \neg q \rceil \quad (7)$$

Statement (5) says that the automaton reacts in less than 2ε seconds to inputs which force a change if there is no stability required for q . Note that less than ε seconds are needed to finish the current cycle and ε seconds are needed to react to this input in the worst case. Formula (6) states this behaviour after $S_t(q)$ seconds: if $S_t(q)$ seconds have elapsed the automaton reacts to inputs which force a change in less than 2ε seconds. In case we know that the automaton has just changed the state then we want to be able to exploit the information that within the next ε seconds another reaction to the inputs in A has to occur. This is formalised by (7).

In Fig. 9 a history is given where these predicates can be used to get information about the behaviour. Statement (5) requires an interval where the state q is constant and no delay requirement is given, i.e. $S_t(q)=0$. If within this interval only inputs were valid which cause a state change, then (5) implies that the length of the interval is shorter than 2ε . In the figure we can apply this formula to the interval $[t0, t1]$ and get the information that it cannot be longer than 2ε .

For the application of (6) we need an interval where the state is q (with $S_t(q) > 0$) only and the length of the interval is longer than $S_t(q)$ seconds. Thus, we can apply this formula to the interval $[t1, t3]$ because the state is T only and the length is longer than 5 s. Since only inputs were true within $[t2, t3]$ which force a state change, we know by (6) that $t3 - t2 < 2\varepsilon$ has to hold.

For (7), an interval of length ε seconds is required where the state is q (with $S_t(q)=0$) and that the state has just changed before the beginning of the interval. Then we know that the state must change just after the interval again if there was no input a during the interval with $q = \delta(q, a)$. The interval $[t3, t4]$ fulfils these requirements. Hence, we know that after $t4$ the state has to change. Note that there is no restriction given by (7) on which succeeding states are allowed. We apply (3) to get more information.

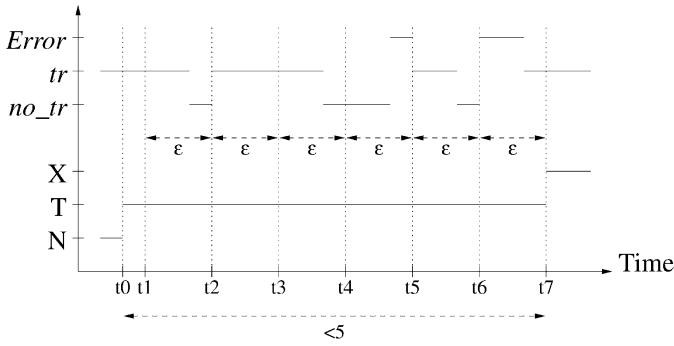


Fig. 10.

Next, we want to describe the automaton's behaviour if it is in a state q where a stability phase is required and the $S_t(q)$ seconds have not elapsed. Then we want to hold this state provided that during this phase only inputs in $S_e(q)$ are read. That means inputs in $S_e(q)$ cannot cause a change of state during the first $S_t(q)$ seconds:

$$S_t(q) > 0 \Rightarrow [\neg q]; [q \wedge A] \xrightarrow{\leq S_t(q)} [q \vee \delta(q, A \setminus S_e(q))] \quad (8)$$

However, we have to take into account the cyclic behaviour of the hardware again. In particular, we should require that if q is left during the stability phase then there has to be an input not contained in $S_e(q)$ at most ε seconds ago:

$$S_t(q) > 0 \Rightarrow [\neg q]; [q]; [q \wedge A] \xrightarrow{\varepsilon \leq S_t(q)} [q \vee \delta(q, A \setminus S_e(q))] \quad (9)$$

Fig. 10 presents a history of the PLC-automaton in Fig. 4 where (8) and (9) are applicable. To apply these formulae we need a change of the state into q with $S_t(q) > 0$. This happens at t_0 where the automaton enters T with $S_t(T) = 5$.

Statement (8) is applicable to all time points t' less than 5 s later than t_0 where the state was constant between $[t_0, t']$. The result of the application is that the state remains in q after t' or changes to a state $\delta(q, a)$ after t' where a is an input that has held somewhere between t_0 and t' and is *not* contained in $S_e(q)$. Thus, we can apply (8) to all intervals of the form $[t_0, t_i]$ with $i \in \{1, \dots, 7\}$. For $1 \leq i \leq 4$ we get the information that no change of the state after t_i is allowed. If $5 \leq i \leq 7$ only a change to X is allowed after t_i .

Formula (9) is applicable to all time points t' less than 5 s later than t_0 and more than ε later than t_0 . It requires the state to be constant between t_0 and t' . The result of the application is that the state remains in q after t' or changes to a state $\delta(q, a)$ after t' where a is an input that has held somewhere between $t' - \varepsilon$ and t' and is *not* contained in $S_e(q)$. Hence, we cannot apply (9) to the interval $[t_0, t_1]$ since the difference between t_0 and t_1 is less than ε seconds. For $i \in \{2, \dots, 7\}$ the difference is big enough and the application yields the following: After t_2 , t_3 , t_4 , and t_6 the state has to remain in T

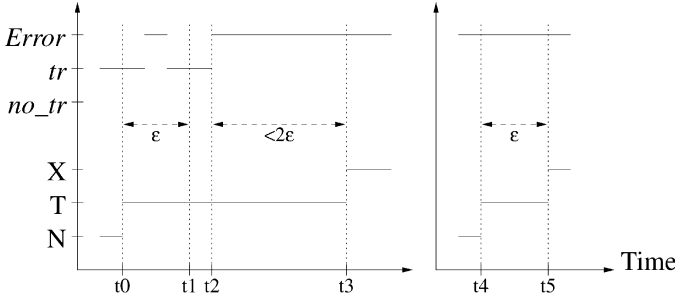


Fig. 11.

after the corresponding time point, and after $t5$ and $t7$ the state is allowed to remain in T or to switch to X after the corresponding time point.

Furthermore, we know that the automaton reacts according to the input if there is a set A that is valid for the last 2ε seconds and disjoint from $S_e(q)$:

$$S_t(q) > 0 \wedge A \cap S_e(q) = \emptyset \wedge q \notin \delta(q, A) \Rightarrow \Box([q \wedge A] \Rightarrow \ell < 2\varepsilon) \quad (10)$$

$$S_t(q) > 0 \wedge A \cap S_e(q) = \emptyset \wedge q \notin \delta(q, A) \Rightarrow [\neg q]; [q \wedge A]^\varepsilon \rightarrow [\neg q] \quad (11)$$

Note that in contrast to (6) predicate (10) does not require that the delay time has elapsed. We demonstrate the meaning of these formulae by the interpretations given in Fig. 11. The application of (10) requires an interval where a state q has held and no input contained in $S_e(q)$ or with a loop from q to q was valid. Then the interval has to be shorter than 2ε . In the left diagram of Fig. 11 there is one interval where (10) is applicable: $[t2, t3]$. Hence, we can derive $t3 - t2 < 2\varepsilon$.

To apply (11) one needs a time point t where the state changes to a state q with $S_t(q) > 0$ and where the state remains in q for the next ε seconds and no input in $S_e(q)$ is valid in that phase. Then the state has to leave q at $t + \varepsilon$. We can apply this to the right diagram of Fig. 11, because at $t4$ the state changes to T and in the following ε seconds this state is held and the input is never no_tr or tr . Hence, the state must leave T at $t5 = t4 + \varepsilon$. Note that (11) is not applicable to the interval $[t0, t1]$ because an input in $S_e(T)$ was valid during that interval.

Formulae (3), (7)–(9), and (11) require a change from $[\neg q]$ to $[q]$ to restrict the possible behaviour. But for the initial state there is no change and therefore the assertions are not applicable in this case. This can be expressed by five corresponding assertions suitable for the initial state; these are given in Appendix A.

Finally, the relation between the observables $\mathcal{S}_{\mathcal{A}}$ and $\mathcal{O}_{\mathcal{A}}$ is established by

$$\Box([q] \Rightarrow [\omega(q)]) \quad (12)$$

This formula says that for each interval $I \leq \text{Time} \subseteq \text{Time}$ the observable $\mathcal{O}_{\mathcal{A}}$ is at time point $t \in I$ equal to $\omega(\mathcal{S}_{\mathcal{A}}(t))$ except for single points.

6. Implementing PLC-automata on PLCs

In this section we want to describe how the PLC-automata can easily be implemented in PLCs. To this end we use the standardised language “ST” (*structured text* [19, 24, 20]) that provides all usual basic constructs of imperative languages and that is used in practise for programming PLCs. We illustrate its usage by means of an example. Let $\mathcal{A} = (Q, \Sigma, \delta, q_0, \varepsilon, S_e, S_t, \Omega, \omega)$ be a PLC-Automaton. Without loss of generality, we assume $Q = \{1, \dots, n\}$, $\Sigma = \{1, \dots, m\}$, and $q_0 = 1$. Then the behaviour of it can be implemented by the ST-program in Fig. 12.

For all three cases it is shown what the PLC has to do. If $S_t(q) = 0$ for a state q , it just has to poll the input and act accordingly ($* \text{ state} = i *$). Otherwise it has to call the timer with the corresponding time value $S_t(q)$ ($* \text{ state} = k *$). Setting the parameter IN to TRUE makes the timer start running for PT seconds if it has not started already. The next statement reads the output Q of the timer. The latter is TRUE iff the time since the starting of the timer has not exceeded PT. By negating this output and storing the result in `time_up` we have a flag that is true iff the time is up. Thus, the first two statements for the case `state=k` in the listing start the timer if needed and register whether the stability time is over or not. Now the PLC has to check the input. If it is an input that is not in $S_e(k)$ ($* u \notin S_e(k) *$) the PLC changes the variable `state` accordingly and in the case of a state change stops the timer by calling it with IN set to FALSE. Otherwise ($* v \in S_e(k) *$) it does the same provided that the time is over. Finally, the output is computed. This ST-program is executed once in each cycle of the PLC. So it is the body of an implicit loop-forever statement.

7. A progress theorem for PLC-automata

This section demonstrates one major advantage of using PLC-automata for specifying controllers: the semantics given in Section 5 allow formal reasoning in DC leading to flexible theorems. Just to give an idea of what can be formally established we state the following theorem and demonstrate its usefulness.

This theorem provides information on how long it takes at most to reach a certain set of states. Often it is necessary that the controller enters a set of states Π' provided that a special set A of inputs holds. E.g. we built the PLC-automaton in Fig. 4 such that the set $\{X\}$ of states is entered provided the set $\{Error\}$ of inputs holds. Usually, the controller is therefore specified in such a way that it will reach Π' after several transitions. Provided that $\Pi' \supseteq \delta^n(Q, A)$ for some $n \in \mathbb{N}_0$ the theorem below estimates the delay until Π' is reached in the worst case. That means that the theorem will give us an upper time bound for the PLC-automaton in Fig. 4 to reach state “X” when reading input “Error” because $\{X\} = \delta^1(\{T, N, X\}, \{Error\})$.


```

VAR state  : INT :=1;
    timer   : TP; (* Timer Type *)
    time_up : BOOL := FALSE;
END_VAR

CASE state OF
  :
  i:      (* state = i, no stability required *)
        state:=  $\delta(i, \text{input})$ ;
        (* end of state=i *)

  :
  k:      (* state = k, stability required *)
        timer(IN:=TRUE, PT:= $t\#S_i(k)$ );
        time_up:=NOT timer.Q;
        CASE input OF
          :
          u:      (*  $u \notin S_e(k)$  *)
                state:=  $\delta(k, u)$ ;
                IF state <> k THEN
                  timer(IN:=FALSE, PT:= $t\#S_i(k)$ );
                END_IF;

          :
          v:      (*  $v \in S_e(k)$  *)
                IF time_up THEN
                  state:=  $\delta(k, v)$ ;
                  IF state <> k THEN
                    timer(IN:=FALSE, PT:= $t\#S_i(k)$ );
                  END_IF;
                END_IF;

          :
        END_CASE;
        (* end of state=k *)

  :
END_CASE;
output:=  $\omega(\text{state})$ ;

```

Fig. 12. The ST-program.

We denote by $\delta^n(\Pi, A)$ the set of states that can be reached by n transitions with an input in set A starting in a state contained in Π . This is inductively defined by

$$\delta^0(\Pi, A) \stackrel{\text{df}}{=} \Pi$$

$$\delta^{n+1}(\Pi, A) \stackrel{\text{df}}{=} \{\delta(q, a) \mid q \in \delta^n(\Pi, A), a \in A\} \text{ for all } n \in \mathbb{N}_0$$

Theorem 4. *Let $\mathcal{A} = (Q, \Sigma, \delta, q_0, \varepsilon, S_e, S_t, \Omega, \omega)$ be a PLC-automaton and let $\Pi \subseteq Q$ and $A \subseteq \Sigma$ with $\delta(\Pi, A) \subseteq \Pi$. Then we have for all $n \in \mathbb{N}_0$:*

$$[\Pi \wedge A] \xrightarrow{c_n} [\delta^n(\Pi, A)] \quad (13)$$

with

$$c_n \stackrel{\text{df}}{=} \varepsilon + \max \left\{ \sum_{i=1}^k s(\pi_i, A) \mid \begin{array}{l} k \leq n \wedge \\ \exists \pi_1, \dots, \pi_k \in \Pi \setminus \delta^n(\Pi, A): \\ \forall 1 \leq j < k: \pi_{j+1} \in \delta(\pi_j, A) \end{array} \right\} \quad (14)$$

where

$$s(\pi, A) \stackrel{\text{df}}{=} \begin{cases} S_t(\pi) + 2\varepsilon & \text{if } S_t(\pi) > 0 \wedge A \cap S_e(\pi) \neq \emptyset \\ \varepsilon, & \text{otherwise} \end{cases} \quad (15)$$

The proof of this theorem can be found in Appendix B. We can apply Theorem 4 to the automaton in Fig. 4 and get, for example, the assertions below:

$$[\{N, T\} \wedge no_tr] \xrightarrow{5+3\varepsilon} [N]$$

$$[\{N, T, X\} \wedge Error] \xrightarrow{2\varepsilon} [X]$$

$$[T \wedge tr] \xrightarrow{\varepsilon} [T]$$

The first assertion can be gained from Theorem 4 by application with $\Pi = \{N, T\}$, $A = \{no_tr\}$, and $n = 1$. It states that if the PLC-automaton is not in state X and reads just no_tr -inputs then the system will be in state N in at most $5 + 3\varepsilon$ seconds. The second assertion is a result of the theorem with $\Pi = \{N, T, X\}$, $A = \{Error\}$, and $n = 1$. It says that the automaton will switch to state X whenever the input $Error$ has held for 2ε seconds. The last assertion uses $\Pi = \{T\}$, $A = \{tr\}$, and $n = 0$. It assures that the PLC-automaton remains in T if during the last ε seconds only the input tr has held.

8. A case study

The following case study illustrates how fast and efficiently real-time systems can be specified and implemented by PLC-automata in comparison with the conventional ProCoS-style. To this end we choose the gas burner case study of the ProCoS-project

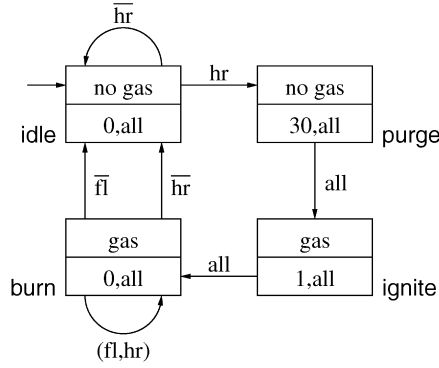


Fig. 13. The gas burner as PLC-automaton.

[6] introduced in Section 4. In the ProCoS-project this case-study was transformed first from Duration Calculus into a certain subset of Duration Calculus called *Implementables*. This yielded a specification of a controller using four states (id (“idle”), pg (“purge”), ig (“ignite”), bn (“burn”)) fulfilling the following assertions:

$$[\Box \vee [\text{id}]; \text{true}] \quad (16)$$

$$[\text{id}] \rightarrow [\text{id} \vee \text{pg}] \quad [\text{pg}] \rightarrow [\text{pg} \vee \text{ig}] \quad (17)$$

$$[\text{ig}] \rightarrow [\text{ig} \vee \text{bn}] \quad [\text{bn}] \rightarrow [\text{bn} \vee \text{id}] \quad (18)$$

$$[\neg \text{pg}]; [\text{pg}] \xrightarrow{\leq 30} [\text{pg}] \quad [\neg \text{ig}]; [\text{ig}] \xrightarrow{\leq 1} [\text{ig}] \quad (19)$$

$$[\text{pg}] \xrightarrow{30+\varepsilon'} [\neg \text{pg}] \quad [\text{ig}] \xrightarrow{1+\varepsilon'} [\neg \text{ig}] \quad (20)$$

$$[\neg \text{id}]; [\text{id} \wedge \neg \text{hr}] \rightarrow [\text{id}] \quad [\neg \text{bn}]; [\text{bn} \wedge \text{hr} \wedge \text{fl}] \rightarrow [\text{bn}] \quad (21)$$

$$[\text{id} \wedge \text{hr}] \xrightarrow{\varepsilon'} [\neg \text{id}] \quad [\text{bn} \wedge \neg \text{hr}] \xrightarrow{\varepsilon'} [\neg \text{bn}] \quad (22)$$

$$[\text{bn} \wedge \neg \text{fl}] \xrightarrow{\varepsilon'} [\neg \text{bn}] \quad (23)$$

The gas valve should be opened iff the state is in $\{\text{bn}, \text{ig}\}$.

In the ProCoS-project this specification was transformed via several steps and interfacing languages to hardware [27, 30, 26]. Due to the special suitability of PLCs to control real-time systems we need not perform these transformations here. It is sufficient to read the specification given above as a specification of a PLC-automaton, which leads us to the PLC-automaton in Fig. 13.

The semantics of the PLC-automaton in this figure refines the specification given in (16)–(23) in the sense that the semantics of the PLC-automaton implies the specification logically. Table 1 shows which assertion of the semantics fulfils the requirements of the specification. The only assumption made is that for the cycle time ε of the PLC-automata the inequality $2\varepsilon \leq \varepsilon'$ holds. This tells the implementor how fast the PLC has

Table 1

Requirement	Is refined by semantic clause
(16)	(2)
(17)–(18)	(3) with $A = \Sigma$
(19)	(8) with $A = \Sigma$
(20)	(6) with $A = \Sigma$ and assuming that $2\varepsilon \leq \varepsilon'$
(21)	(3) with $A = \{\neg \text{hr}\}$ resp. $A = \{\text{hr} \wedge \text{fl}\}$
(22)–(23)	(5) with $A = \{\text{hr}\}$, $\{\neg \text{hr}\}$, or $A = \{\neg \text{fl}\}$

to cycle in the worst case. And this problem normally corresponds to the question: “How much money do we have to spend for the hardware in order to guarantee that the upper time bound is not violated?”

In [9, 11] the interested reader can find an algorithm that generalises the result that we can find a PLC-automaton that refines the specification in terms of Implementables. This algorithm synthesises a PLC-automaton from a specification using Implementables. This algorithm works provided that the specification does not contain contradictory constraints. Moreover, in [9, 11] it is shown that the algorithm produces correct results in the sense that the semantics of the synthesised PLC-automaton refines the given specification.

9. Parallelism and PLC-automata

In the previous sections we introduced a formalism that reflects the intuition and daily practice of engineers who implement real-time controllers. In this section we define different parallel composition operators for PLC-automata motivated by the different manifestations of parallelism for PLCs. Roughly speaking, the parallel composition of PLC-automata can be represented by the conjunction of the semantics of each PLC-automata. But there are three phenomena which can change the character of the parallel composition of PLC-automata (cf. Fig. 14):

Transmission: Depending on the transmission medium between two PLCs the behaviour of both can vary. The media can introduce transmission delays or errors. To get a provably correct system one has to model the actual transmission medium in a semantically adequate manner.

Pipelining: The input of one PLC-automaton can be the output of another PLC-automaton. If both automata are implemented on the same PLC, it is possible to describe the allowed behaviour in more detail.

Synchronisation: Suppose two PLC-automata implemented on the same PLC share an input. Depending on the construction of both automata it may be possible that a certain combination of states is not reachable due to the synchronisation on the shared input. The semantics of the parallel composition should be strong enough to establish such behaviour.

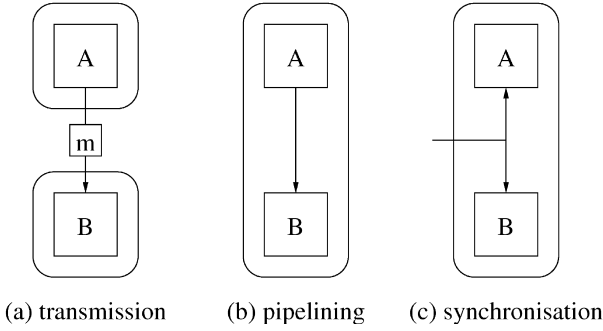


Fig. 14. Parallel compositions of PLC-automata.

9.1. Transmission

We present a uniform approach to model transmission of information between different PLCs. Basically, transmission between two PLC-automata is a relation between the output-observable of the first automaton and the input-observable of the second one. We describe this relation by DC-formulae speaking about both observables.

Suppose that the output of PLC-automaton \mathcal{A} is the input of PLC-automaton \mathcal{B} via the medium m . We denote this connection with the following symbol:

$$\mathcal{A} \overset{m}{\gg} \mathcal{B}.$$

The semantics of $\mathcal{A} \overset{m}{\gg} \mathcal{B}$ is defined as follows:

$$\llbracket \mathcal{A} \overset{m}{\gg} \mathcal{B} \rrbracket \stackrel{\text{df}}{=} \llbracket \mathcal{A} \rrbracket_{\text{DC}} \wedge \llbracket \mathcal{B} \rrbracket_{\text{DC}} \wedge \llbracket m \rrbracket_{\mathcal{O}_{\mathcal{A}}}^{\mathcal{I}_{\mathcal{B}}},$$

where $\llbracket m \rrbracket_{\mathcal{O}_{\mathcal{A}}}^{\mathcal{I}_{\mathcal{B}}}$ denotes a relation between $\mathcal{O}_{\mathcal{A}}$ and $\mathcal{I}_{\mathcal{B}}$.

Note that this definition of transmission is not very restrictive. It is possible to interpret a PLC-automaton as a medium because it is a relation between its input and output. Typically, we are interested in the delay time of the transmission when we deal with real-time systems. Hence, we define a standard medium sm that is parameterised by the delay time t and define a relation between the observables of type R , i.e., $\mathcal{I}, \mathcal{O} : \text{Time} \rightarrow R$, as follows:

$$\begin{aligned} \llbracket sm(t) \rrbracket_{\mathcal{I}}^{\mathcal{O}} \stackrel{\text{df}}{=} & \forall I : \emptyset \neq I \subseteq R \Rightarrow [\mathcal{I} \in I] \xrightarrow{t} [\mathcal{O} \in I] \\ & \wedge \neg([\mathcal{I} \in I]^{<t}; [\mathcal{O} \notin I]; \text{true}) \end{aligned}$$

Informally speaking, the possible outputs of $sm(t)$ at time $t_0 \in \text{Time}$ are the inputs that were valid during $] \max(0, t_0 - t), t_0[$. We use $\mathcal{A} \overset{t}{\gg} \mathcal{B}$ as an abbreviation for $\mathcal{A} \overset{sm(t)}{\gg} \mathcal{B}$.

9.2. Pipelining

Unlike transmission, pipelining assumes that we consider two PLC-automata that are implemented on the same PLC. In principle, pipelining could be modelled as an

“internal transmission” of data between the automata in the same way as in Section 9.1, but we would loose information that results from the common implementation.

In the pipelining case we know that the result computed by the first automaton during a cycle is used in the same cycle by the second automaton as input. That means every output of the first automaton will be read by the second one. If both automata change state in the same cycle, the external observer will notice these changes simultaneously. To model this we have to use more than the two observables of the transmission case. Hence, we are not able to express pipelining as a special case of transmission.

Transmission from a PLC-automaton \mathcal{A} to automaton \mathcal{B} requires the source-code of both automata to be organised as follows:

- The declaration part has to contain uniquely named variables for both automata.
- The body of \mathcal{A} has to precede the body of \mathcal{B} .
- The output of \mathcal{A} has to be a subset of the input of \mathcal{B} .
- In the body of \mathcal{B} the input-variable has to be replaced by the name of the output-variable of \mathcal{A} .

Because of the similarity of pipelining and transmission we use $\mathcal{A} \gg \mathcal{B}$ to denote a pipelining from automaton \mathcal{A} to \mathcal{B} . The semantics of $\mathcal{A} \gg \mathcal{B}$ is given by

$$\llbracket \mathcal{A} \gg \mathcal{B} \rrbracket \stackrel{\text{df}}{=} \llbracket \mathcal{A} \rrbracket_{\text{DC}} \wedge \llbracket \mathcal{B} \rrbracket_{\text{DC}} \wedge \text{pipe}(\mathcal{A}, \mathcal{B}),$$

where $\text{pipe}(\mathcal{A}, \mathcal{B})$ is the conjunction of the following formulae ranging over all states $q \in Q_{\mathcal{A}}$ and all $q' \in Q_{\mathcal{B}}$ and all sets A with $\emptyset \neq A \subseteq \Sigma_{\mathcal{A}}$. Read ε as $\min(\varepsilon_{\mathcal{A}}, \varepsilon_{\mathcal{B}})$.

$$\llbracket \neg(q \wedge q') \rrbracket; \llbracket q \wedge q' \wedge A \rrbracket \rightarrow \left[q \wedge q' \vee \bigvee_{a \in A} \delta_{\mathcal{A}}(q, a) \wedge \delta_{\mathcal{B}}(q', \omega_{\mathcal{A}}(\delta_{\mathcal{A}}(q, a))) \right] \quad (23a)$$

$$\vee \bigvee_{a \in A \cap S_{\varepsilon, \mathcal{A}}(q), S_{t, \mathcal{A}}(q) > 0} q \wedge \delta_{\mathcal{B}}(q', \omega_{\mathcal{A}}(q)) \quad (23b)$$

$$\vee \bigvee_{a \in A, \omega_{\mathcal{A}}(\delta_{\mathcal{A}}(q, a)) \in S_{\varepsilon, \mathcal{B}}(q'), S_{t, \mathcal{B}}(q') > 0} \delta_{\mathcal{A}}(q, a) \wedge q' \quad (23c)$$

$$\llbracket q \wedge q' \wedge A \rrbracket \xrightarrow{\varepsilon} \left[q \wedge q' \vee \bigvee_{a \in A} \delta_{\mathcal{A}}(q, a) \wedge \delta_{\mathcal{B}}(q', \omega_{\mathcal{A}}(\delta_{\mathcal{A}}(q, a))) \right] \quad (24a)$$

$$\vee \bigvee_{a \in A \cap S_{\varepsilon, \mathcal{A}}(q), S_{t, \mathcal{A}}(q) > 0} q \wedge \delta_{\mathcal{B}}(q', \omega_{\mathcal{A}}(q)) \quad (24b)$$

$$\vee \bigvee_{a \in A, \omega_{\mathcal{A}}(\delta_{\mathcal{A}}(q, a)) \in S_{\varepsilon, \mathcal{B}}(q'), S_{t, \mathcal{B}}(q') > 0} \delta_{\mathcal{A}}(q, a) \wedge q' \quad (24c)$$

This pair is similar to (3) and (4) but they restrict the progress of both automata involved. In (23a) and (24a) we allow simultaneous steps. In (23b) and (24b) we allow a change of the second automaton without a change of the first one provided

that an input is read for which a delay is valid. Eqs. (23c) and (24c) allow steps of the first component without a change of the second one provided that the new state of the first one should be delayed.

Note that the formulae above allow nonsimultaneous steps even if the delay times have elapsed. Hence, we need further formulae to disallow this kind of behaviour. Suppose $S_{t,\mathcal{A}}(q) > 0$:

$$\left([q]^{S_{t,\mathcal{A}}(q)+2\varepsilon} \wedge \text{true}; \left(\bigvee [\neg q']; [q' \wedge A] \right) \right) \rightarrow \left[q \wedge q' \vee \bigvee_{a \in A} \delta_{\mathcal{A}}(q, a) \wedge \delta_{\mathcal{B}}(q', \omega_{\mathcal{A}}(\delta_{\mathcal{A}}(q, a))) \right] \quad (25a)$$

$$\bigvee_{a \in A, \omega_{\mathcal{A}}(\delta_{\mathcal{A}}(q, a)) \in S_{e,\mathcal{B}}(q'), S_{t,\mathcal{B}}(q') > 0} \left[\delta_{\mathcal{A}}(q, a) \wedge q' \right] \quad (25b)$$

Note that the RHS of this formula consists of the combinations of states given in (23a) and (23c). Similarly, the case of $S_{t,\mathcal{B}}(q') > 0$:

$$\left([q']^{S_{t,\mathcal{B}}(q')+2\varepsilon} \wedge \text{true}; \left(\bigvee [\neg q]; [q \wedge A] \right) \right) \rightarrow \left[q \wedge q' \vee \bigvee_{a \in A} \delta_{\mathcal{A}}(q, a) \wedge \delta_{\mathcal{B}}(q', \omega_{\mathcal{A}}(\delta_{\mathcal{A}}(q, a))) \right] \quad (26a)$$

$$\bigvee_{a \in A \cap S_{e,\mathcal{A}}(q), S_{t,\mathcal{A}}(q) > 0} \left[q \wedge \delta_{\mathcal{B}}(q', \omega_{\mathcal{A}}(q)) \right] \quad (26b)$$

9.3. Synchronisation

In the case of automata that are implemented on the same PLC and share the input we can benefit from the knowledge that during each cycle the same input is read by both automata. Thus it is often the case that certain combination of states are not reachable in such a synchronised system. We enhance our semantics by predicates that allow us to establish such phenomena.

To this end, we define the semantics of two PLC-automata \mathcal{A} and \mathcal{B} which share the input-observable $\mathcal{I} = \mathcal{I}_{\mathcal{A}} = \mathcal{I}_{\mathcal{B}}$ (in symbols: $\mathcal{A} \parallel_{\mathcal{I}} \mathcal{B}$ or simply $\mathcal{A} \parallel \mathcal{B}$ if \mathcal{I} is clear) in a similar way as before:

$$\llbracket \mathcal{A} \parallel_{\mathcal{I}} \mathcal{B} \rrbracket \stackrel{\text{df}}{=} \llbracket \mathcal{A} \rrbracket_{\text{DC}} \wedge \llbracket \mathcal{B} \rrbracket_{\text{DC}} \wedge \text{syn}(\mathcal{A}, \mathcal{B})$$

where $\text{syn}(\mathcal{A}, \mathcal{B})$ is the conjunction of the following formulae ranging over all states $q \in Q_{\mathcal{A}}$ and all $q' \in Q_{\mathcal{B}}$ and all nonempty sets A that are subsets of the range of \mathcal{I} .

Again, read ε as $\min(\varepsilon_{\mathcal{A}}, \varepsilon_{\mathcal{B}})$. Note that these formulae are defined in analogy to the formulae for the semantics of pipelining:

$$\begin{aligned} \lceil \neg(q \wedge q') \rceil; \lceil q \wedge q' \wedge A \rceil \rightarrow & \left[q \wedge q' \vee \bigvee_{a \in A} \delta_{\mathcal{A}}(q, a) \wedge \delta_{\mathcal{B}}(q', a) \right. \\ & \vee \bigvee_{a \in A \cap S_{e, \mathcal{A}}(q), S_{t, \mathcal{A}}(q) > 0} q \wedge \delta_{\mathcal{B}}(q', a) \\ & \left. \vee \bigvee_{a \in A \cap S_{e, \mathcal{B}}(q'), S_{t, \mathcal{B}}(q') > 0} \delta_{\mathcal{A}}(q, a) \wedge q' \right] \end{aligned} \quad (27)$$

$$\begin{aligned} \lceil q \wedge q' \wedge A \rceil \xrightarrow{\varepsilon} & \left[q \wedge q' \vee \bigvee_{a \in A} \delta_{\mathcal{A}}(q, a) \wedge \delta_{\mathcal{B}}(q', a) \right. \\ & \vee \bigvee_{a \in A \cap S_{e, \mathcal{A}}(q), S_{t, \mathcal{A}}(q) > 0} q \wedge \delta_{\mathcal{B}}(q', a) \\ & \left. \vee \bigvee_{a \in A \cap S_{e, \mathcal{B}}(q'), S_{t, \mathcal{B}}(q') > 0} \delta_{\mathcal{A}}(q, a) \wedge q' \right] \end{aligned}$$

If $S_{t, \mathcal{A}}(q) > 0$ holds we have:

$$\begin{aligned} & \left(\lceil q \rceil^{S_{t, \mathcal{A}}(q) + 2\varepsilon} \wedge \text{true}; \left(\bigvee \lceil \neg q' \rceil; \lceil q' \wedge A \rceil^\varepsilon \right) \right) \\ & \rightarrow \left[q \wedge q' \vee \bigvee_{a \in A} \delta_{\mathcal{A}}(q, a) \wedge \delta_{\mathcal{B}}(q', a) \vee \bigvee_{a \in A \cap S_{e, \mathcal{B}}(q'), S_{t, \mathcal{B}}(q') > 0} \delta_{\mathcal{A}}(q, a) \wedge q' \right] \end{aligned}$$

In the case of $S_{t, \mathcal{B}}(q') > 0$ we add:

$$\begin{aligned} & \left(\lceil q \rceil^{S_{t, \mathcal{B}}(q') + 2\varepsilon} \wedge \text{true}; \left(\bigvee \lceil \neg q \rceil; \lceil q \wedge A \rceil^\varepsilon \right) \right) \\ & \rightarrow \left[q \wedge q' \vee \bigvee_{a \in A} \delta_{\mathcal{A}}(q, a) \wedge \delta_{\mathcal{B}}(q', a) \vee \bigvee_{a \in A \cap S_{e, \mathcal{A}}(q), S_{t, \mathcal{A}}(q) > 0} q \wedge \delta_{\mathcal{B}}(q', a) \right] \end{aligned}$$

9.4. Examples of parallel composition

In this subsection we present examples of the three parallel operators given before and use of the additional semantics formulae for each case. Consider the gasburner in Fig. 13 again. This automaton produces a boolean signal whether the gas valve should be opened or not. Assume now that we have to produce an output not only for the actuator of the gas valve but also for the ignition. To this end we change the automaton of Fig. 13 in such a way that the internal state becomes the output. The result is given

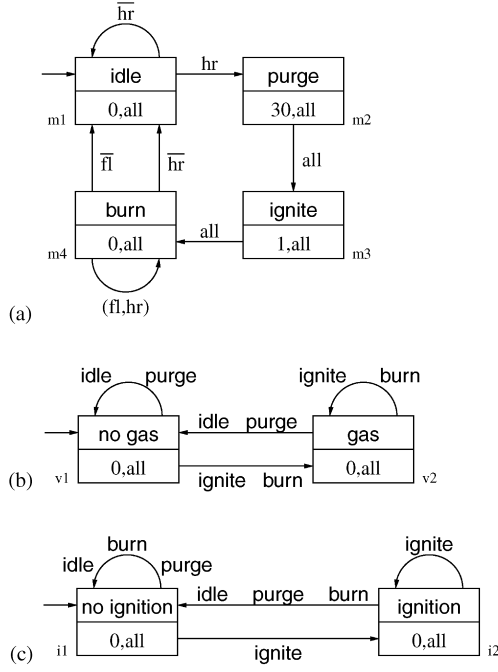


Fig. 15. Gasburner components with ignition.

in Fig. 15 by automaton \mathcal{A} . Furthermore, we define an automaton \mathcal{B} that computes the output gas iff \mathcal{A} outputs ig or bn and an automaton \mathcal{C} that computes the output ignition iff \mathcal{A} outputs ig.

Consider the system $\mathcal{B} \parallel \mathcal{C}$. From a common implementation on one PLC we expect that ignition only occurs simultaneously with gas. From the set of DC-formulae of $\text{syn}(\mathcal{B}, \mathcal{C})$ we can use (27) with $A = \{\text{id}, \text{pg}, \text{ig}, \text{bn}\}$ to prove our expectation:

$$\begin{aligned}
 & [\neg(\mathbf{v1} \wedge \mathbf{i1})]; [\mathbf{v1} \wedge \mathbf{i1}] \rightarrow [(\mathbf{v1} \wedge \mathbf{i1})] \vee \underbrace{(\mathbf{v2} \wedge \mathbf{i2})}_{\text{ig}} \vee \underbrace{(\mathbf{v2} \wedge \mathbf{i1})}_{\text{bn}} \\
 & [\neg(\mathbf{v2} \wedge \mathbf{i2})]; [\mathbf{v2} \wedge \mathbf{i2}] \rightarrow [(\mathbf{v2} \wedge \mathbf{i2})] \vee \underbrace{(\mathbf{v1} \wedge \mathbf{i1})}_{\text{id,pg}} \vee \underbrace{(\mathbf{v2} \wedge \mathbf{i1})}_{\text{bn}} \\
 & [\neg(\mathbf{v2} \wedge \mathbf{i1})]; [\mathbf{v2} \wedge \mathbf{i1}] \rightarrow [(\mathbf{v2} \wedge \mathbf{i1})] \vee \underbrace{(\mathbf{v1} \wedge \mathbf{i1})}_{\text{id,pg}} \vee \underbrace{(\mathbf{v2} \wedge \mathbf{i2})}_{\text{ig}}
 \end{aligned}$$

These formulae prove that the synchronised system will never enter the combination $\mathbf{v1} \wedge \mathbf{i2}$ which corresponds due to (12) to ignition without gas. We start in $\mathbf{v1} \wedge \mathbf{i1}$. The

first formula says that the system can only switch to $\mathbf{v2} \wedge \mathbf{i1}$ or $\mathbf{v2} \vee \mathbf{i2}$. Similarly, the following formulae assure the exclusion of a change to the critical state.

If we implement \mathcal{A} and \mathcal{B} on the same PLC it is reasonable to implement \mathcal{A} first, i.e. $\mathcal{A} \gg \mathcal{B}$. If we analyse $\text{pipe}(\mathcal{A}, \mathcal{B})$ in the same way as $\text{syn}(\mathcal{B}, \mathcal{C})$ before we can find the following properties for the whole system:

$$\begin{aligned}
 & [\neg(\mathbf{m1} \wedge \mathbf{v1})]; [\mathbf{m1} \wedge \mathbf{v1}] \rightarrow [(\mathbf{m1} \wedge \mathbf{v1}) \vee \underbrace{(\mathbf{m2} \wedge \mathbf{v1})}_{\text{hr}}] \\
 & [\neg(\mathbf{m2} \wedge \mathbf{v1})]; [\mathbf{m2} \wedge \mathbf{v1}] \rightarrow [(\mathbf{m2} \wedge \mathbf{v1}) \vee \underbrace{(\mathbf{m3} \wedge \mathbf{v2})}_{\text{all}}] \\
 & [\neg(\mathbf{m3} \wedge \mathbf{v2})]; [\mathbf{m3} \wedge \mathbf{v2}] \rightarrow [(\mathbf{m3} \wedge \mathbf{v2}) \vee \underbrace{(\mathbf{m4} \wedge \mathbf{v2})}_{\text{all}}] \\
 & [\neg(\mathbf{m4} \wedge \mathbf{v2})]; [\mathbf{m4} \wedge \mathbf{v2}] \rightarrow [(\mathbf{m4} \wedge \mathbf{v2}) \vee \underbrace{(\mathbf{m1} \wedge \mathbf{v1})}_{\neg\text{fl} \vee \neg\text{hr}}]
 \end{aligned}$$

From these formulae we can conclude with some simple DC-arguments that $\mathbf{v2}$ holds iff \mathcal{A} is in $\mathbf{m3}$ or $\mathbf{m4}$. Due to the definition of the output of \mathcal{B} we know that gas holds iff $\mathbf{v2}$ is true. Hence, from the observer's point of view the PLC-Automaton of Fig. 13 and $\mathcal{A} \gg \mathcal{B}$ are equivalent.

Consider now a system in which \mathcal{A} and \mathcal{B} are implemented on distinct PLCs. We assume that the transmission medium is the standard medium $sm(\Delta)$ with an arbitrary $\Delta > 0$. We are interested in the delay between the reactions of \mathcal{A} and \mathcal{B} that is introduced by the transmission. Assume that the cycle time of \mathcal{B} is $\varepsilon_{\mathcal{B}}$. By the semantics of $\mathcal{A} \xrightarrow{\Delta} \mathcal{B}$ we know that the following holds:

$$\begin{aligned}
 & [\mathcal{O}_{\mathcal{A}} \in \{\text{id}, \text{pg}\}] \xrightarrow{\Delta} [\mathcal{I}_{\mathcal{B}} \in \{\text{id}, \text{pg}\}] \\
 & [\mathcal{O}_{\mathcal{A}} \in \{\text{ig}, \text{bn}\}] \xrightarrow{\Delta} [\mathcal{I}_{\mathcal{B}} \in \{\text{ig}, \text{bn}\}]
 \end{aligned}$$

By Theorem 4 applied to \mathcal{B} with all states and both sets of inputs above we get the following assertions:

$$\begin{aligned}
 & [\mathcal{S}_{\mathcal{B}} \in \{\mathbf{v1}, \mathbf{v2}\} \wedge \mathcal{I}_{\mathcal{B}} \in \{\text{id}, \text{pg}\}] \xrightarrow{2\varepsilon_{\mathcal{B}}} [\mathbf{v1}] \\
 & [\mathcal{S}_{\mathcal{B}} \in \{\mathbf{v1}, \mathbf{v2}\} \wedge \mathcal{I}_{\mathcal{B}} \in \{\text{ig}, \text{pg}\}] \xrightarrow{2\varepsilon_{\mathcal{B}}} [\mathbf{v2}]
 \end{aligned}$$

Note that $\mathcal{S}_{\mathcal{B}} \in \{\mathbf{v1}, \mathbf{v2}\}$ is equivalent to true. Because $[P_1] \xrightarrow{\varepsilon_1} [P_2]$ and $[P_2] \xrightarrow{\varepsilon_2} [P_3]$ implies $[P_1] \xrightarrow{\varepsilon_1 + \varepsilon_2} [P_3]$ we can summarise these to

$$\begin{aligned}
 & [\mathcal{O}_{\mathcal{A}} \in \{\text{id}, \text{pg}\}] \xrightarrow{\Delta + 2\varepsilon_{\mathcal{B}}} [\mathbf{v1}] \\
 & [\mathcal{O}_{\mathcal{A}} \in \{\text{ig}, \text{bn}\}] \xrightarrow{\Delta + 2\varepsilon_{\mathcal{B}}} [\mathbf{v2}]
 \end{aligned}$$

That means that the worst-case delay of the gas signal is $\Delta + 2\varepsilon_{\mathcal{B}}$.

10. A timed automaton approach

In the following, we give an operational semantics of a PLC-automaton in terms of a set of timed traces accepted by a timed automaton. In [13, 14] it was proven that the following semantics is stronger than the DC semantics given in Section 5 (and equivalent to a slight extension of that semantics). For the definition of timed automata the reader is referred to [2, 25].

Definition 5. Let $\mathcal{A} = (Q, \Sigma, \delta, q_0, \varepsilon, S_t, S_e, \Omega, \omega)$ be a PLC-automaton. We define $\mathcal{T}(\mathcal{A}) \stackrel{\text{df}}{=} (\mathcal{S}, \mathcal{X}, \mathcal{L}, \mathcal{E}, \mathcal{I}, \mathcal{P}, \mu, S_0)$ with

- $\mathcal{S} \stackrel{\text{df}}{=} \{0, 1, 2, 3\} \times \Sigma \times \Sigma \times Q$ as locations,
- $\mathcal{X} \stackrel{\text{df}}{=} \{x, y, z\}$ as clocks,
- $\mathcal{L} \stackrel{\text{df}}{=} \Sigma \cup \{\text{poll}, \text{test}, \text{tick}\}$ as labels,
- The set of transitions \mathcal{E} consists of the following transitions, for each $i \in \{0, 1, 2, 3\}$, $a, b, c \in \Sigma$, and $q \in Q$,

$$(i, a, b, q) \xrightarrow{c, \text{true}, \{x\}} (i, c, b, q) \quad \text{if } c \neq a \quad (28)$$

$$(0, a, b, q) \xrightarrow{\text{poll}, 0 < x \wedge 0 < z, \emptyset} (1, a, a, q) \quad (29)$$

$$(1, a, b, q) \xrightarrow{\text{test}, y \leq S_t(q), \emptyset} (2, a, b, q) \quad \text{if } S_t(q) > 0 \wedge b \in S_e(q) \quad (30)$$

$$(1, a, b, q) \xrightarrow{\text{test}, y > S_t(q), \emptyset} (3, a, b, q) \quad \text{if } S_t(q) > 0 \wedge b \in S_e(q) \quad (31)$$

$$(1, a, b, q) \xrightarrow{\text{test}, \text{true}, \emptyset} (3, a, b, q) \quad \text{if } S_t(q) = 0 \vee b \notin S_e(q) \quad (32)$$

$$(2, a, b, q) \xrightarrow{\text{tick}, \text{true}, \{z\}} (0, a, b, q) \quad (33)$$

$$(3, a, b, q) \xrightarrow{\text{tick}, \text{true}, \{z\}} (0, a, b, \delta(q, b)) \quad \text{if } q = \delta(q, b) \quad (34)$$

$$(3, a, b, q) \xrightarrow{\text{tick}, \text{true}, \{y, z\}} (0, a, b, \delta(q, b)) \quad \text{if } q \neq \delta(q, b) \quad (35)$$

- $\mathcal{I}(s) \stackrel{\text{df}}{=} z \leq \varepsilon$ as invariant for each location $s \in \mathcal{S}$,
- $\mathcal{P} = \Sigma \cup Q \cup \Omega$ is the set of propositions,
- $\mu(i, a, b, q) \stackrel{\text{df}}{=} a \wedge q \wedge \omega(q)$ as propositions for each location $(i, a, b, q) \in \mathcal{S}$, and
- $S_0 \stackrel{\text{df}}{=} \{(0, a, b, q_0) \mid a, b \in \Sigma\}$ as set of initial locations.

The set of locations¹ of $\mathcal{T}(\mathcal{A})$ (refer to Definition 5) consists of four dimensions. The first part (“program counter”) with range $\{0, 1, 2, 3\}$ describes the internal status of the polling system. “0” denotes the first part of the cycle. The polling has not occurred yet. “1” denotes that the polling has happened in the current cycle. The check whether

¹ Note that “locations” refers to the timed automaton and “states” to the PLC-automaton.

to react has not occurred yet. “2” denotes that polling and testing have happened. The system decided not to react to the input. “3” denotes that polling and testing have happened. The system decided to react to the input. The second component of the locations denotes the latest input event while the third component contains the latest polled input. The last component represents the current state of the PLC-automaton. There are three clocks in use: Clock x measures how long the latest input is valid, clock y measures how long the current state is valid, and clock z measures the time of the current cycle. Transitions that change the first component of the locations are labelled with *poll*, *test*, and *tick*. The remaining transitions (28) are labelled with inputs and are not restricted anyhow. They change the second component which represents the latest input-event. The third component describes the input which is polled by the system. The polling has to happen after an amount of time since the beginning of the cycle. To this end the clock z is used. This clock denotes the elapsed time for the current cycle. Hence, the polling transition (29) is labelled with the condition $z > 0$. Furthermore, it is not allowed to poll an input at the same time point where it gets valid. Hence, we introduced the clock x denoting the time since the last input is valid and restricted the *poll*-event with $x > 0$. Otherwise the system could react to input that was valid only for a point of time. After the polling the testing has to occur (30)–(32). These transitions reflect the decision of the system whether to react to the polled b input or not. It depends on the definitions of S_e , S_t , and the value of the y -clock which denotes the time how long the current *state* q is valid. It can only decide to ignore the input when $b \in S_e(q)$ and $S_t(q) > 0$ are true (30) and moreover the delay time has not elapsed: $y \leq S_t(q)$. Finally, the *tick*-events finish the cycle (33)–(35). Depending on the previous decision by the *test*-event the state may change or not. All necessary clocks are reset. Due to the invariants $z \leq \varepsilon$ for all locations we know that a cycle consisting of a *poll*-, a *test*-, and a *tick*-event has to happen within ε seconds because only the *tick*-event resets z .

11. Concluding remarks

Tool support is indispensable for the development of correct software. In [31, 32] the reader can find the description of a tool supporting software development with PLC-automata. It allows to edit PLC-automata with hierarchical extensions as defined in [15], i.e. PLC-automata with mechanisms to structure a design in a way similar to StateCharts [17, 18]. With this tool the user can build networks of PLC-automata, simulate these networks, perform some static timing analysis, and translate them into input for the Model-Checkers Uppaal [4] and Kronos [7]. Furthermore, we look for systematic ways to develop PLC-Automata from specifications. A first result of this research is presented in [9, 11], where a synthesis algorithm is presented for the subset of DC-formulae called Implementables (cf. Section 8).

We made also comprehensive case studies to evaluate our approach: Academic case studies like the gas burner (Section 8), the Production Cell [23, 22], and the

audio-protocol [5] as well as case studies of industrial complexity like a redesign of a traffic control system for tramways with complicated driving rules which was originally provided by the industrial partner of the UniForM-project.

Acknowledgements

I would like to thank E.-R. Olderog and all other members of the “semantics group” in Oldenburg for detailed comments and various discussions on the subject of this paper. Furthermore, I would like to thank H. Becker for enhancing the readability of this paper.

Appendix A. Additional semantics formulae for the start

In Section 5 we presented the duration calculus semantics for a PLC-automaton. Some of the formulae given there require a change of state to restrict the behaviour of the system. Hence, they are not applicable for the initial state in the initial phase. Nevertheless, we expect the system to behave as if there had been a change of state. To formalise this expectation we add the following constraints which restrict only the start of the system. Each of them correspond to a formula in Section 5 which is expressed by the enumeration:

- (3') $\neg(\lceil q_0 \wedge A \rceil; \lceil \neg(q_0 \vee \delta(q_0, A)) \rceil; \text{true})$
- (7') $S_t(q_0) = 0 \wedge q_0 \notin \delta(q_0, A) \Rightarrow \neg(\lceil q_0 \wedge A \rceil^e; \lceil q_0 \rceil; \text{true})$
- (8') $S_t(q_0) > 0 \Rightarrow \neg(\lceil q_0 \wedge A \rceil^{<S_t(q_0)}; \lceil \neg(q_0 \vee \delta(q_0, A \setminus S_e(q_0))) \rceil; \text{true})$
- (9') $S_t(q_0) > 0 \Rightarrow \neg(\lceil \lceil q_0 \rceil; \lceil q_0 \wedge A \rceil^e \rceil^{<S_t(q_0)}; \lceil \neg(q_0 \vee \delta(q_0, A \setminus S_e(q_0))) \rceil; \text{true})$
- (11') $S_t(q_0) > 0 \wedge A \cap S_e(q_0) = \emptyset \wedge q_0 \notin \delta(q_0, A) \Rightarrow \neg(\lceil q_0 \wedge A \rceil^e; \lceil q_0 \rceil; \text{true})$

Appendix B. Proof of Theorem 4

Proof. The proof is by contradiction. We use the following properties which are easy to prove with $\delta(\Pi, A) \subseteq \Pi$:

$$\forall k, j \in \mathbb{N}_0: \delta^{k+j}(\Pi, A) \subseteq \delta^k(\Pi, A)$$

Assume the negation of (13):

$$\begin{aligned} & \neg(\lceil \Pi \wedge A \rceil \xrightarrow{c_n} \lceil \delta^n(\Pi, A) \rceil) \\ \Leftrightarrow & \neg(\neg(\text{true}; \lceil \Pi \wedge A \rceil^{c_n}; \lceil \neg\delta^n(\Pi, A) \rceil; \text{true})) \\ \Leftrightarrow & \text{true}; \lceil \Pi \wedge A \rceil^{c_n}; \lceil \neg\delta^n(\Pi, A) \rceil; \text{true} \end{aligned}$$

Due to the finite variability of \mathcal{S} we can split the second interval into finitely many subintervals where only one state in Π occurs.

$$\Rightarrow \exists m \in \mathbb{N}_0, \pi_0, \dots, \pi_m \in \Pi: \forall 0 \leq i < m: \pi_i \neq \pi_{i+1} \\ \wedge \text{true}; ([A]^{c_n} \wedge [\pi_0]; \dots; [\pi_m]); [\neg \delta^n(\Pi, A)]; \text{true}$$

Applying (3) to π_i with $i = 1, \dots, m-1$ and expanding the abbreviations yields $\neg(\text{true}; [\neg \pi_i]; [\pi_i \wedge A]; [\neg(\pi_i \vee \delta(\pi_i, A))]); \text{true}$). That means, if only inputs in A could be read since the change to state π_i the state can change only to a state contained in $\delta(\pi_i, A)$. Hence, $\pi_{i+1} \in \delta(\pi_i, A)$ holds.

$$\Rightarrow \exists m \in \mathbb{N}_0, \pi_0, \dots, \pi_m \in \Pi: \forall 0 \leq i < m: \pi_i \neq \pi_{i+1} \\ \wedge \text{true}; ([A]^{c_n} \wedge [\pi_0]; \dots; [\pi_m]); [\neg \delta^n(\Pi, A)]; \text{true} \\ \wedge \forall i \in \{2, \dots, m\}: \pi_i \in \delta^{i-1}(\pi_1, A)$$

If $m > 0$ we can apply the same argument to get $\pi_m \notin \delta^n(\Pi, A)$: Assume that $\pi_m \in \delta^n(\Pi, A)$ holds. From (3) we can conclude that after the $[\pi_m]$ -phase only states in $\Gamma \stackrel{\text{df}}{=} \delta(\pi_m, A) \cup \{\pi_m\}$ are allowed. With the assumption and $\delta(\Pi, A) \subseteq \Pi$ it is the case that $\Gamma \subseteq \delta^n(\Pi, A)$ which contradicts the requirement that after the $[\pi_m]$ -phase a $[\neg \delta^n(\Pi, A)]$ -phase follows. In the case of $m = 0$ we can use (4) to get the same result because from (14) we know that $c_n \geq \varepsilon$. Hence, we get

$$\Rightarrow \exists m \in \mathbb{N}_0, \pi_0, \dots, \pi_m \in \Pi: \forall 0 \leq i < m: \pi_i \neq \pi_{i+1} \\ \wedge \text{true}; ([A]^{c_n} \wedge [\pi_0]; \dots; [\pi_m]); [\neg \delta^n(\Pi, A)]; \text{true} \\ \wedge \pi_m \notin \delta^n(\Pi, A) \wedge \forall i \in \{2, \dots, m\}: \pi_i \in \delta^{i-1}(\pi_1, A)$$

Because of $\delta(\Pi, A) \subseteq \Pi$ it is not possible that $\pi_i \in \delta(\pi_i, A)$ holds for an $i \in \{1, \dots, m\}$. Otherwise this would contradict $\pi_m \notin \delta^n(\Pi, A)$. Furthermore, $m \leq n$ must hold: If $m > n$ we have $\pi_m \in \delta^{m-1}(\pi_1, A)$ which implies $\pi_m \in \delta^n(\Pi, A)$.

$$\Rightarrow \exists m \in \{0, \dots, n\}, \pi_0, \dots, \pi_m \in \Pi: \forall 0 \leq i < m: \pi_i \neq \pi_{i+1} \\ \wedge \text{true}; ([A]^{c_n} \wedge [\pi_0]; \dots; [\pi_m]); [\neg \delta^n(\Pi, A)]; \text{true} \\ \wedge \pi_m \notin \delta^n(\Pi, A) \wedge \forall i \in \{2, \dots, m\}: \pi_i \in \delta^{i-1}(\pi_1, A) \\ \wedge \forall i \in \{1, \dots, m\}: \pi_i \notin \delta(\pi_i, A)$$

We are now able to derive upper time bounds for the $[\pi_i]$ -intervals with $i \geq 1$. If $S_t(\pi_i) = 0$ or $S_t(\pi_i) > 0 \wedge A \cap S_e(\pi_i) = \emptyset$ we can use (7) and (11) resp. which give us the bound of ε seconds. In the case of $S_t(\pi_i) > 0 \wedge A \cap S_e(\pi_i) \neq \emptyset$ we get the bound $S_t(\pi_i) + 2\varepsilon$ by (6). Thus we have the following:

$$\Rightarrow \exists m \in \{0, \dots, n\}, \pi_0, \dots, \pi_m \in \Pi: \forall 0 \leq i < m: \pi_i \neq \pi_{i+1} \\ \wedge \text{true}; ([A]^{c_n} \wedge [\pi_0]; [\pi_1]^{\leq s(\pi_1, A)}; \dots; [\pi_m]^{\leq s(\pi_m, A)}); \\ [\neg \delta^n(\Pi, A)]; \text{true}$$

$$\begin{aligned} & \wedge \pi_m \notin \delta^n(\Pi, A) \wedge \forall i \in \{2, \dots, m\}: \pi_i \in \delta^{i-1}(\pi_1, A) \\ & \wedge \forall i \in \{1, \dots, m\}: \pi_i \notin \delta(\pi_i, A) \end{aligned}$$

Let us now consider two cases: either the length of the $\lceil \pi_0 \rceil$ -interval is less than ε or not:

$$\begin{aligned} \Rightarrow \exists m \in \{0, \dots, n\}, \pi_0, \dots, \pi_m \in \Pi: & \forall 0 \leq i < m: \pi_i \neq \pi_{i+1} \\ & \wedge \text{true}; (\lceil A \rceil^{c_n} \wedge \lceil \pi_0 \rceil^{<\varepsilon}; \lceil \pi_1 \rceil^{\leq s(\pi_1, A)}; \dots; \lceil \pi_m \rceil^{\leq s(\pi_m, A)}); \\ & \quad \lceil \neg \delta^n(\Pi, A) \rceil; \text{true} \\ & \wedge \pi_m \notin \delta^n(\Pi, A) \wedge \forall i \in \{2, \dots, m\}: \pi_i \in \delta^{i-1}(\pi_1, A) \\ & \wedge \forall i \in \{1, \dots, m\}: \pi_i \notin \delta(\pi_i, A) \\ \vee \exists m \in \{0, \dots, n\}, \pi_0, \dots, \pi_m \in \Pi: & \forall 0 \leq i < m: \pi_i \neq \pi_{i+1} \\ & \wedge \text{true}; (\lceil A \rceil^{c_n} \wedge \lceil \pi_0 \rceil^{\geq \varepsilon}; \lceil \pi_1 \rceil^{\leq s(\pi_1, A)}; \dots; \lceil \pi_m \rceil^{\leq s(\pi_m, A)}); \\ & \quad \lceil \neg \delta^n(\Pi, A) \rceil; \text{true} \\ & \wedge \pi_m \notin \delta^n(\Pi, A) \wedge \forall i \in \{2, \dots, m\}: \pi_i \in \delta^{i-1}(\pi_1, A) \\ & \wedge \forall i \in \{1, \dots, m\}: \pi_i \notin \delta(\pi_i, A) \end{aligned}$$

The first case is a contradiction because c_n is the length of the $\lceil A \rceil$ -interval and the accumulated upper time bounds for the $\lceil \pi_i \rceil$ require a length less than

$$\varepsilon + \sum_{i=1}^m s(\pi_i, A).$$

Due to the definition of c_n both cannot hold. In the second case we know from (4) that $\pi_1 \in \delta(\pi_0, A)$. Hence, $m = n$ is not possible due to $\pi_m \in \delta^{m-1}(\pi_1, A)$, thus $\pi_m \in \delta^m(\pi_0, A)$ but $\pi_m \notin \delta^n(\Pi, A)$.

\Rightarrow false

$$\begin{aligned} \vee \exists m \in \{0, \dots, n-1\}, \pi_0, \dots, \pi_m \in \Pi: & \forall 0 \leq i < m: \pi_i \neq \pi_{i+1} \\ & \wedge \text{true}; (\lceil A \rceil^{c_n} \wedge \lceil \pi_0 \rceil^{\geq \varepsilon}; \lceil \pi_1 \rceil^{\leq s(\pi_1, A)}; \dots; \lceil \pi_m \rceil^{\leq s(\pi_m, A)}); \\ & \quad \lceil \neg \delta^n(\Pi, A) \rceil; \text{true} \\ & \wedge \pi_m \notin \delta^n(\Pi, A) \wedge \forall i \in \{1, \dots, m\}: \pi_i \in \delta^i(\pi_0, A) \\ & \wedge \forall i \in \{0, \dots, m\}: \pi_i \notin \delta(\pi_i, A) \end{aligned}$$

We derive upper time bounds for the $\lceil \pi_0 \rceil$ -interval. If $S_t(\pi_0) = 0$ or $S_t(\pi_0) > 0 \wedge A \cap S_e(\pi_0) = \emptyset$ we can use (5) and (10) resp. giving us the bound of $< 2\varepsilon$ seconds. In the case of $S_t(\pi_0) > 0 \wedge A \cap S_e(\pi_0) \neq \emptyset$ we get the bound $< S_t(\pi_0) + 2\varepsilon$ by (6) which can be weakened to $< s(\pi_0, A) + \varepsilon$. Thus we have the following:

$$\begin{aligned} \Rightarrow \exists m \in \{0, \dots, n-1\}, \pi_0, \dots, \pi_m \in \Pi: & \forall 0 \leq i < m: \pi_i \neq \pi_{i+1} \\ & \wedge \text{true}; (\lceil A \rceil^{c_n} \wedge \lceil \pi_0 \rceil^{<s(\pi_0, A)+\varepsilon}; \lceil \pi_1 \rceil^{\leq s(\pi_1, A)}; \dots; \lceil \pi_m \rceil^{\leq s(\pi_m, A)}); \end{aligned}$$

$$\begin{aligned}
& [\neg \delta^n(\Pi, A)]; \text{true} \\
& \wedge \pi_m \notin \delta^n(\Pi, A) \wedge \forall i \in \{1, \dots, m\}: \pi_i \in \delta^i(\pi_0, A) \\
& \wedge \forall i \in \{0, \dots, m\}: \pi_i \notin \delta(\pi_i, A)
\end{aligned}$$

This is a contradiction as in the previous case, because the accumulated upper time bounds for the $[\pi_i]$ require a length less than

$$\varepsilon + \sum_{i=0}^m s(\pi_i, A).$$

Due to the definition of c_n this is not possible.

\Rightarrow false \square

References

- [1] R. Alur, C. Courcoubetis, D. Dill, Model-checking for real-time systems, 5th Annu. IEEE Symp. on Logic in Computer Science, IEEE Press, New York, 1990, pp. 414–425.
- [2] R. Alur, D.L. Dill, A theory of timed automata, Theoret. Comput. Sci. 126 (1994) 183–235.
- [3] R. Alur, T. Henzinger, E. Sontag (Eds.), in: Hybrid Systems III, Lecture Notes in Computer Science, vol. 1066, Springer, Berlin, 1996.
- [4] J. Bengtsson, K.G. Larsen, F. Larsson, P. Pettersson, Wang Yi, Uppaal – a tool suite for automatic verification of real-time systems, in: R. Alur, T. Henzinger, E. Sontag (Eds.), Hybrid Systems III, Lecture Notes in Computer Science, vol. 3, Springer, Berlin, 1996, pp. 232–243.
- [5] D. Bosscher, I. Polak, F. Vaandrager, Verification of an audio control protocol, in: H. Langmaack, W.-P. de Roever, J. Vytöpil (Eds.), Formal Techniques in Real-Time and Fault-Tolerant Systems, Lecture Notes in Computer Science, vol. 863, Springer, Berlin, 1994, pp. 170–192.
- [6] J. Bowen, C.A.R. Hoare, H. Langmaack, E.-R. Olderog, A.P. Ravn, ProCoS II: A ProCoS II Project Final Report, Chapter 3 Number 59 in Bulletin of the EATCS, European Association for Theoretical Computer Science, June 1996, pp. 76–99.
- [7] C. Daws, A. Olivero, S. Tripakis, S. Yovine, The tool Kronos, in: R. Alur, T. Henzinger, E. Sontag (Eds.), Hybrid Systems III, Lecture Notes in Computer Science, vol. 3, Springer, Berlin, 1996, pp. 208–219.
- [8] H. Dierks, PLC-Automata: a new class of implementable real-time automata, in: M. Bertran, T. Rus (Eds.), ARTS'97, Lecture Notes in Computer Science, Mallorca, Spain, vol. 1231, Springer, Berlin, May 1997, pp. 111–125.
- [9] H. Dierks, Synthesising controllers from real-time specifications, in: 10th Internat. Symp. on System Synthesis, IEEE Computer Society, New York, September 1997, pp. 126–133, short version of [11].
- [10] H. Dierks, Comparing model-checking and logical reasoning for real-time systems, in: Workshop Proc. of the ESSL'98, 1998, pp. 13–22.
- [11] H. Dierks, Synthesizing controllers from real-time specifications, IEEE Transac. Comput.-Aided Design Integrated Circuits Systems 18 (1) (1999) 33–43.
- [12] H. Dierks, C. Dietz, Graphical specification and reasoning: Case study generalized railroad crossing, in: J. Fitzgerald, C.B. Jones, P. Lucas (Eds.), FME'97, Lecture Notes in Computer Science, vol. 1313, Graz, Austria, Springer, Berlin, September 1997, pp. 20–39.
- [13] H. Dierks, A. Fehnker, A. Mader, F.W. Vaandrager, Operational and logical semantics for polling real-time systems, in: Ravn, Rischel (Eds.), Formal Techniques in Real-Time and Fault-Tolerant Systems, Lecture Notes in Computer Science, vol. 1486, Lyngby, Denmark, Springer, Berlin, September 1998, pp. 29–40, short version of [14].
- [14] H. Dierks, A. Fehnker, A. Mader, F.W. Vaandrager, Operational and logical semantics for polling real-time systems, Technical Report CSI-R9813, Computer Science Institute Nijmegen, Faculty of Mathematics and Informatics, Catholic University of Nijmegen, April 1998, full paper of [13].

- [15] H. Dierks, J. Tapken, Tool-supported hierarchical design of distributed real-time systems, in: Proc. 10th EuroMicro Workshop on Real Time Systems, IEEE Computer Society, June 1998, pp. 222–229.
- [16] M.R. Hansen, Zhou Chaochen, Duration calculus: logical foundations, *Formal Aspects Comput.* 9 (1997) 283–330.
- [17] D. Harel, Statecharts: a visual formalism for complex systems, *Sci. Comput. Programming* 8 (1987) 231–274.
- [18] D. Harel, On visual formalisms, *Comm. ACM* 31 (5) (May 1988) 514–530.
- [19] IEC International Standard 1131-3, Programmable Controllers, Part 3, Programming Languages, 1993.
- [20] K.-H. John, M. Tiegkamp, SPS-Programmierung Mit IEC 1131-3, Springer, Berlin, 1995 (in German).
- [21] B. Krieg-Brückner, J. Peleska, E.-R. Olderog, D. Balzer, A. Baer, UniForM – universal formal methods workbench, in: U. Grote, G. Wolf (Eds.), Statusseminar des BMBF Softwaretechnologie, BMBF, Berlin, March 1996, pp. 357–378.
- [22] C. Lewerentz (Ed.), Formal Development of Reactive Systems: Case Study “Production Cell”, Lecture Notes in Computer Science, vol. 891, Springer, Berlin, 1995.
- [23] C. Lewerentz, T. Lindner (Eds.), Case Study “Production Cell”, Forschungszentrum Informatik, Karlsruhe, 1994.
- [24] R.W. Lewis, Programming industrial control systems using IEC 1131-3, The Institution of Electrical Engineers, 1995.
- [25] O. Maler, S. Yovine, Hardware timing verification using kronos, in: Proc. 7th Conf. on Computer-based Systems and Software Engineering, IEEE Press, New York, 1996.
- [26] M. Müller-Olm, Modular Compiler Verification, Lecture Notes in Computer Science, vol. 1283, Springer, Berlin, 1997.
- [27] A.P. Ravn, Design of embedded real-time computing systems, Technical Report 1995-170, Technical University of Denmark, 1995.
- [28] A.P. Ravn, H. Rischel (Eds.), Formal Techniques in Real-Time and Fault-Tolerant Systems, Lecture Notes in Computer Science, vol. 1486, Lyngby, Denmark, Springer, Berlin, September 1998.
- [29] A.P. Ravn, H. Rischel, K.M. Hansen, Specifying and verifying requirements of real-time systems, *IEEE Trans. Software Eng.* 19 (1993) 41–55.
- [30] M. Schenke, Development of correct real-time systems by refinement, Habilitation Thesis, University of Oldenburg, April 1997.
- [31] J. Tapken, Interactive and compilative simulation of PLC-Automata, in: W. Hahn, A. Lehmann (Eds.), ESS’97, SCS, October 1997, pp. 552–556.
- [32] J. Tapken, H. Dierks, MOBY/PLC – Graphical Development of PLC-Automata, in: A.P. Ravn, H. Rischel (Eds.), Formal Techniques in Real-Time and Fault-Tolerant Systems, Lecture Notes in Computer Science, vol. 1486, Lyngby, Denmark, Springer, Berlin, September 1998, pp. 311–314.
- [33] Zhou Chaochen, C.A.R. Hoare, A.P. Ravn, A calculus of durations, *Inform. Proc. Lett.* 40/5 (1991) 269–276.